Security Lab – Public File Storage Service

Virtual Machine

This lab can be done with the **Ubuntu VM**. The remainder of this document assumes you are working with this VM.

The lab could basically also be done on any Linux system with an installed IDE, Java and nmap. However, it is possible to render a system non-usable by carrying out the attacks in this lab (especially attack 1 in section 6). It's therefore strongly recommended you do this lab using the Ubuntu VM.

1 Introduction

In this lab, you get the server-side component of a very simple file storage service that is implemented in Java as a basis. Basically, the service allows to store and fetch files, and to execute system commands. The service doesn't use any kind of login, i.e., every user can easily store and fetch files and users can also read and overwrite each other's files. Therefore, the service is intended only for temporary storage of non-sensitive data, e.g., to easily exchange files between two devices of the same user or of different users, like a super simple version of Dropbox. Despite its simplicity, the server component contains several serious vulnerabilities. Your task is to find and understand the vulnerabilities and to correct the code such that the vulnerabilities are no longer present.

One goal of this lab is to «sharpen your awareness» in the sense that you should understand how much can go wrong with respect to security even in a simple program if the developer lacks a profound understanding of software security. Such developers also often assume the user (or attacker) uses standard client software (here: the client component of the file storage service, but you won't use this in this lab) that follows the protocol correctly and only sends well-formed data. But in reality, attackers use specialized tools to arbitrarily interact with a server application, which may allow them to do attacks the developer didn't consider during development.

This lab focuses in particular on input validation and what can happen if input validation is neglected. A significant portion of security vulnerabilities that show up in the wild are related to a lack of input validation and this lab will show you various attacks that can happen in such cases and will also teach you some fundamental techniques to do input validation correctly. The second goal of this lab is therefore to get a good understanding about vulnerabilities and solution approaches in the context of input validation.

2 Basis for this Lab

- Download *PublicFileStorage.zip* from Moodle.
- Move the file to an appropriate location (e.g., into a directory *securitylabs* in the home directory */home/user*).
- Unzip the file. The resulting directory *PublicFileStorage* contains a Java project based on Maven. This should be importable in any modern IDE. The remainder assumes you are using *NetBeans*, which is installed on the Ubuntu image.
- Start *NetBeans* and open the project.
- To build the project, right-click *PublicFileStorage* in the *Projects* tab and select *Build*.
    - Whenever you make changes and save them, the project should be automatically rebuilt. However, this does not work reliably and therefore, it's recommended that you always select *Build* again before running the program.
    - The executable code is placed in directory *PublicFileStorage/target/classes*.
- The project contains two programs:
    - *PublicFileStorageServer.java*: The secure file storage server component.

- *PublicFileStorageTester.java*: A test program to perform different attacks against the server. You can look at the code, but please don't change it! The program uses two command line parameters: The host name or the IP address of the server and a number (0-8) of the attack you want to execute (see sections 4 and 6). Entering no number at all runs all attacks.
- Both programs are located in package *ch.zhaw.securitylab.publicfilestorage*, which means the class files are placed in directory *ch/zhaw/securitylab/publicfilestorage* below *PublicFileStorage/target/classes.*
- In addition, a file *Common.java* is included. This includes constants that are used by both programs.

- The server uses the directory *files* to store the uploaded files. When starting the server, two files *file1.txt* and *file2.txt* are already present in this directory. The directory *files* is located below *PublicFileStorage/target/classes*.

3 Functionality and Protocol

Communication between client and server uses a simple, ASCII-based protocol. Messages from client to server are called *requests*, messages from server to client are called *responses*. Overall, three different request types are supported: *GET*, *PUT* and *SYSTEM*. In the following, the request types and how they are used in the communication protocol are described.

3.1 GET

GET is used to fetch a file. To get *file1.txt*, the following request is sent to the server:

```
GET file1.txt
-----DONE-----
```

When the server receives this, the requested file is accessed (in this case *files/file1.txt*, and its content is sent back to the client as follows:

```
OK
-----CONTENT-----
This is the content of file1.txt.
It even contains a second line.
And a third as well, unbelievable!
-----DONE-----
```

The file content is included line by line between -----*CONTENT*----- and -----*DONE*-----. Note that for simplicity, we assume the files are always text files as this makes it much simpler to print the file content in a terminal (see later). Security-wise, supporting binary files wouldn't change anything. If a problem happens while processing the request, e.g., if the file does not exist, a *NOK* response is sent back:

```
NOK
-----DONE-----
```

The request and responses above also show some general properties of the protocol used by the file storage service:

- Every request always starts with one of the three request types (*GET, PUT, SYSTEM*), followed by one or more arguments (here one argument: *file1.txt*).
- Every response always starts with a response status *OK* or *NOK*, depending on whether the request could successfully be handled or not.
- Every request and every response is always terminated with a marker -----*DONE*-----.

- Every part of the request or response uses a separate line. Lines are always terminated with a new-line character (*\n*). For instance, the *GET* request to the server contains two lines – *GET file1.txt* and *-----DONE-----* – and both lines are terminated with a *\n* character.
- As an additional feature (although not shown in the messages above), the arguments in the first line of a request may be URL encoded (this is also supported by some other ASCII-based protocols, e.g., HTTP). URL encoding uses a *%* character followed by the hex value of the ASCII code of the character. So for instance, the URL encoded value of *A* is *%41*. As a result of supporting URL encoding, the arguments are URL decoded by the server before they are processed further.

3.2 PUT

PUT is used to store a file on the server. To create a file *testfile.txt* with specific content, the following request is sent to the server:

```
PUT testfile.txt
-----CONTENT-----
Test data: Terrific test file content!
Spread across two lines.
-----DONE-----
```

When the server receives this, the file is created (in this case *files/testfile.txt*) and written with the specified content. If the file already exists, it is overwritten. If the file can be created and (over-) written, the following response is sent back to the client:

```
OK
-----DONE-----
```

If a problem happens while processing the request, the same *NOK* response as with *GET* requests is sent back.

For simplicity, no subdirectories are supported, so files can only be written to (and read) from *files/*, but not to/from, e.g., *files/subdir/*.

3.3 SYSTEM

SYSTEM is used to execute a command in the underlying operating system of the server. The only command that is currently supported is *USAGE* to get the disk space that is consumed by the files on the server. To get the disk space consumed by the files, the following request is sent to the server:

```
SYSTEM USAGE *
-----DONE-----
```

When the server receives this, the server accesses the *du* command in the underlying operating system (using *du -h files/**) and includes the output of the *du* command in the response as follows:

```
OK
-----CONTENT-----
4.0K    files/file1.txt
4.0K    files/file2.txt
-----DONE-----
```

Note that *du* returns the disk usage (i.e., the size of the used disk blocks) and not the effective size of the files, which is why *4.0K* is returned for both files. If a problem happens while processing the request, the same *NOK* response as with *GET* requests is sent back.

Besides *USAGE* *, the request can also use *USAGE .* (with a dot instead of an asterisk), which returns the total disk usage of all the files on the server.

4 Run the Server and Test whether it Works Correctly

First, run the server and test whether it works correctly. To run the server, open a terminal (it's best to always run the programs in a terminal), change to directory *PublicFileStorage/target/classes* and enter:

```
$ java ch.zhaw.securitylab.publicfilestorage.PublicFileStorageServer
```

Per default, the server uses TCP port 4567. If you want to, you can change this in *Common,java*, but there's usually no reason to do so.

To test whether the server works correctly, use a second terminal, change again to directory *PublicFileStorage/target/classes* and enter the command below to run the test program. The command line parameters identify the hostname of the server (*localhost*) and the number of the test or attack that should be executed. Here, we are using test number 0. Numbers 1 and higher are «reserved» for the actual attacks:

```
$ java ch.zhaw.securitylab.publicfilestorage.PublicFileStorageTester
  localhost 0
```

The test consists of several requests to test the various request types. The output you get in the terminal of *PublicFileStorageTester* should be self-explanatory. As an example, test 0a is described:

```
Test 0a: Check GET (file existing)... done
Test **SUCCEEDED**
Status:  OK
Content: This is the content of file1.txt belonging to user1.
It even contains a second line.
And a third as well, unbelievable!

Server still running
```

The first line describes the test. Here, a *GET* request is tested that uses an existing file name. The second line *Test **SUCCEEDED*** means that the test was successful, i.e., it worked as expected. A failure would be indicated with *Test **FAILED***. Next, there's the status of the received response, which is either *OK* or *NOK* – here it is *OK* because the *GET* request could successfully be handled. Next, there's the content of the response, which – in this case – contains the content of the requested file. Finally, it is indicated whether the server is still reachable (*Server still running*) or not (*Server crashed*).

Whenever you are doing some changes to the server code throughout the remainder of this lab, it's always a good idea to re-run this test to check whether the basic legitimate requests are still handled correctly.

5 Source Code Analysis of *PublicFileStorageServer.java*

Before you start solving the tasks in the next section, you should study the source code of the server to get a good understanding about its functionality. Basically, the program works as follows:

- The *main* method creates a *PublicFileStorageServer* object and starts the server (*run* method).
- In the *run* method, a while loop is used to listen for connection requests from clients. When a client establishes a connection, the method *accept* returns and delivers a *Socket* object, which can be used to communicate with the client. This object is passed to the *processRequest* method, which handles the specific request. Note that for simplicity, the server runs single-threaded, which means the next connection (from the same or from another client) will only be accepted once the previous request has been completely handled (including sending back the response).

- *processRequest* reads the first line of the request and checks the type of the request. Depending of this type (*GET*, *PUT* or *SYSTEM*), it calls a corresponding method (*serveFile*, *storeFile* or *executeSystemCommand*), where the remainder of the request is handled and where the response to the client is generated. When *processRequest* is completed, program control goes back to the *run* method so the next connection request can be accepted.

Take your time to get a good understanding of the code – you'll need this understanding later in this lab to make the right program adaptations to fix the security vulnerabilities.

6 Attacks and Countermeasures

In each of the following subsections, *PublicFileStorageTester* is used to carry out an attack against the running server. Your task is always the same: First, carry out the attack to see its effect. Then determine the reason why the attack was successful. Next, adapt the server code to fix the vulnerability (this is needed with all attacks except attack 1). Finally, check whether the attack is truly prevented by executing it again, and also check whether test 0 (see section 4) can still be executed successfully.

It's important that you go through the attacks in the given order as in some cases, the attacks build upon attacks and countermeasures of previous subsections. Also, always make sure to completely solve a subsection before starting with the next one. When adapting the code, you should always focus on fixing the vulnerability that allows the current attack and then you should verify whether the vulnerability has indeed been removed by your corrective measures.

The test program tries to «find out» whether an attack can be carried out or whether your corrective measures prevent the attack – the output of the test program provides you with the necessary information. If the test program generates an exception, then this usually means that you haven't fixed the vulnerability correctly (according to the specifications in the individual subsections).

6.1 Attack 1: Compromising the root account by setting a new root password

In this attack, you are exploiting vulnerabilities in the server to set a new root password in the underlying system. Before you execute the attack, you should create a copy of the shadow file (which contains the passwords of the system users). This requires root permission and can be done – assuming you are working in a terminal as *user* – with the *sudo* command, followed by providing the password of *user* (which is *user*):

```
$ sudo cp /etc/shadow /etc/shadow.org
```

Next, start the server so that it runs with root permissions. This is also done with *sudo*:

```
$ sudo java
  ch.zhaw.securitylab.publicfilestorage.PublicFileStorageServer
```

Finally, execute the attack (which has number 1) by entering the following in another terminal:

```
$ java ch.zhaw.securitylab.publicfilestorage.PublicFileStorageTester
  localhost 1
```

Inspecting the output of the test program should inform you whether the attack was successful.

Now, try to get root access by using *su* in a terminal. Use the password *test*. The login attempt should be successful, which means the root account was indeed compromised by setting a new root password *test* (the original password was *root*) in the shadow file.

This can be verified by comparing the original and modified shadow files, e.g., by using *diff*:

```
$ sudo diff /etc/shadow /etc/shadow.org
```

What has changed? What hasn't changed?

Why are the passwords not directly visible?

The fact that this attack was successful has multiple reasons – which ones? To provide a complete answer to this, you'll have to study the source code of the server, and also the requests (especially the first line of them) that are received by the server (for the latter, there's already a code line in method *processRequest*, but it's currently commented). It may also be helpful using some further debugging messages in the server code (e.g., with *System.out.println()*).

To remove the effect of the attack, replace the modified shadow file again with the original one:

```
$ sudo mv /etc/shadow.org /etc/shadow
```

Next, stop server and restart it, but this time as *user*:

```
$ java ch.zhaw.securitylab.publicfilestorage.PublicFileStorageServer
```

Execute the attack again. What happens? Explain the behavior.

With this, the attack is prevented, because one of the reasons that made it possible has been removed (as the server no longer runs with root permissions). However, the other reasons that made the attack possible (which are vulnerabilities in the code and which you likely identified above) are still there. This will be fixed later.

The important take away message of this attack is that you should never run a program with root privileges (unless this is really necessary), because if there exists a vulnerability, the impact is usually higher when the program runs with root permissions.

In the following, always run the server as *user*.

6.2 Attack 2: DoS attack with empty request

Make sure the server is running and execute attack 2:

```
$ java ch.zhaw.securitylab.publicfilestorage.PublicFileStorageTester
  localhost 2
```

The server should crash. As the server is no longer available for legitimate users, we identify this as a DoS (Denial of Service) attack.

Analyze why the server crashed by studying the request and the source code of the server. The generated exception provides you with hints where in the code you should look. In the following box, explain why the server crashed.

How would you fix this problem? Just think about a possible solution, don't implement anything yet.

Before adapting the code, let's discuss the vulnerability in more detail. As you probably found out above, the problem is that the server tries to process the request, which consists of an empty first line. As a result of this, a *StringIndexOutOfBoundsException* is thrown. As the exception is not caught, the program is terminated. This is a typical example of a vulnerability related to a lack of input validation. Input validation issues are very prevalent in software and more of them will follow in subsequent attacks. Generally speaking, a lack of input validation means that the software (here the server) blindly accepts and processes input data without validating the data first, i.e., without making sure that it conforms to an expected and legitimate data format before processing it. This can then lead to unexpected and unforeseen situations, which can have serious security implications. Here, it «only» resulted in terminating the server and thereby affects availability, but – as you'll see later – it can also have an impact on confidentiality and integrity.

The solution is to first validate input data before processing it. One reasonable way to do this in the current case is to implement a method with name *validateFirstLineOfRequest(String input)*, which gets the first line of the request as a parameter. The method then checks if the first line consists of a valid format. Looking at the protocol definition in section 3, you can see that a valid first request line consists of the request type (*GET, PUT, SYSTEM*), followed by one space character, followed by one or more arguments, whereas each argument consists of printable characters (except space characters) and where the arguments are separated by space characters. This format can best be tested using a regular expression and in general, regular expressions play an important role when implementing input validation rules. Using a regular expression, *validateFirstLineOfRequest* can be implemented as follows:

```
private boolean validateFirstLineOfRequest(String input) {
  if (input != null && input.matches(
      "^(GET|PUT|SYSTEM) [\\x21-\\x7E][\\x20-\\x7E]*$")) {
    return true;
  }
  return false;
}
```

Study this method, especially the regular expression (in bold). In the following box, explain in detail why the regular expression is suited to validate the format of the first line of a request by describing the purpose of the different parts of the regular expression. Note that the actual regular expression contains only single slashes (\), but in a Java string, it's necessary to escape a slash character (using \\) so that it is interpreted as a literal slash character. This means that the substring *[\\x21-\\x7E][\\x20-\\x7E]* corresponds to *[\x21-\x7E][\x20-\x7E]* in the actual regular expression.

Hint: To play around with regular expressions, and to learn what the individual parts of a regular expression mean, *https://regex101.com* is an excellent resource.

Next, call the method above in method *processRequest* to validate the first line of each request before processing it. If validation fails, respond with *NOK*. The easiest way to implement this is as follows (new code in bold):

```
if (line == null) {
  // Apparently, the client disconnected without sending anything, do
     nothing
} else {
  // System.out.println("First line of request: " + line);

  // Validate the first line of the request
  if (!validateFirstLineOfRequest(line)) {
    writeNOKNoContent(toClient);
    return;
  }

  // Get request type and argument from the first line of the request.
  int indexSpace = line.indexOf(' ');
  ...
```

To verify whether your code modifications have the desired effect, execute the attack again and check the output of the test program. The attack should now fail. Also, repeat test 0 to make sure the server still works as it should when the client uses legitimate requests.

6.3 Attack 3: DoS attack with long request

Make sure the server is running and execute attack 3:

```
$ java ch.zhaw.securitylab.publicfilestorage.PublicFileStorageTester
  localhost 3
```

Again, the server should crash. Analyze the exception, the request and the source code of the server and explain why the server crashed.

The client sends a request where the first line of the request has a length of 5 GB. The attack could basically be prevented by assigning the server more memory (in Java, this is possible with the option -Xmx, e.g., -Xmx10G). Why is this a poor solution for the identified problem?

How would you fix this problem? Just think about a possible solution, don't implement anything yet.

To fix this problem, the best approach is to implement a variant of *readLine* that only reads a line up to a certain maximum number of characters. Do this by implementing a method *readLineMaxChar* that uses the following interface:

```
private String readLineMaxChar(Reader reader, int max)
    throws IOException {...}
```

The method gets an object *reader* of type *Reader* (e.g., a *BufferedReader* to read data from a socket) and a maximum number of characters to read (*max*). It then reads a line from *reader* (terminated with \n or end-of-file) and returns it (without the \n). If the line is longer than *max* characters (not counting the \n), an *IOException* is thrown. To prevent an *OutOfMemoryError*, it's important that you count the read characters while reading them character-by-character from *reader* and that you throw an *IOException* as soon as more than *max* characters have been read (the remaining characters will be ignored in this case). If no data can be read from *reader* at all (i.e., if the first read operation returns end-of-file), *readLineMaxChar* should return *null*.

Then, use this method instead of the standard method *readLine* in *processRequest*, i.e., replace

```
// Read the first line of the request from the client
```

```
String line = fromClient.readLine();
```

with

```
// Read the first line of the request from the client
String line;
try {
  line = readLineMaxChar(fromClient, 1000);
} catch (IOException e) {
  writeNOKNoContent(toClient);
  return;
}
```

As you can see, the maximum line length is limited to 1'000 (it's best to use a constant for this). This is certainly good enough to process the first line on any reasonable request. If *readLineMaxChar* throws an exception (which means the line is longer than 1'000 characters), respond with *NOK*, as illustrated in the code above.

To verify whether your code modifications have the desired effect, execute the attack again and check the output of the test program. The attack should now fail. Also, repeat test 0 to make sure the server still works as it should when the client uses legitimate requests.

6.4 Attack 4: DoS attack with PUT request with a long file line

Make sure the server is running and execute attack 4:

```
$ java ch.zhaw.securitylab.publicfilestorage.PublicFileStorageTester
  localhost 4
```

Again, the server should crash. Analyze the exception, the request and the source code of the server and explain why the server crashed.

Having already prevented attack 3, it should be obvious how to prevent attack 4: Make sure that *readLineMaxChar* is used instead of *readLine* for *all* the lines read from the client and not just for the first one. Therefore, adapt the code so that all *readLine* calls where data is read from the client are replaced with *readLineMaxChar* (do this throughout the entire code to make sure that all attacks that use too long lines are prevented). For simplicity, we restrict every single line to 1'000 characters, just like above. If *readLineMaxChar* throws an exception, always respond with *NOK*. Once you have done these adaptations, all code lines of this form:

```
line = fromClient.readLine();
```

should have been replaced with code lines of this form (including the necessary exception handling):

```
line = readLineMaxChar(fromClient, 1000);
```

To verify whether your code modifications have the desired effect, execute the attack again and check the output of the test program. The attack should now fail. Also, repeat test 0 to make sure the server still works as it should when the client uses legitimate requests.

6.5 Attack 5: DoS attack with PUT request with many lines

Make sure the server is running and execute attack 5:

```
$ java ch.zhaw.securitylab.publicfilestorage.PublicFileStorageTester
  localhost 5
```

Again, the server should crash. Analyze the exception, the request and the source code of the server and explain why the server crashed.

Again, it shouldn't be too difficult to identify the problem. Fix the vulnerability in *storeFile* by making sure that a file that should be stored cannot contain more than 1'000 lines. If there are more than 1'000 lines, respond with *NOK*. This means that in combination with the fix introduced in the previous subsection, a file can contain at most 1'000 lines, where each line cannot be longer than 1'000 characters. Note again that these limitations are chosen quite arbitrarily and in a real application, you'd probably choose other bounds; but what's important is that you realize that you have to specify and enforce some upper bound for the file size as otherwise, DoS conditions will likely occur.

To verify whether your code modifications have the desired effect, execute the attack again and check the output of the test program. The attack should now fail. Also, repeat test 0 to make sure the server still works as it should when the client uses legitimate requests.

With this attack, we are stepping away from DoS attacks that target the availability. Maybe you found attacks 2-5 a bit repetitive as the attacks are all somewhat similar in the sense that they triggered different exceptions in the server that all resulted in terminating the running program. But exactly this is an important lesson that you should take away from attacks 2-5: In many cases, there are different ways to achieve an attack goal (here: four different ways to crash the server) and the attacker just needs to find one of these ways to be successful. So if you prevent three of the attacks by fixing the associated vulnerabilities but don't fix the fourth vulnerability, the attacker still wins if he finds the remaining vulnerability. Therefore, when thinking about possible attacks against the system you are developing (which is a very important activity during a secure software development process), you should always consider many different ways how a specific attack goal can be achieved and then you should make sure that all these ways are prevented in the code. In this sense, attacks 2-5 certainly gave you a taste what it means to prevent all different ways to achieve an attack goal and to truly achieve secure software.

6.6 Attack 6: Getting a file that shouldn’t be accessible (variant 1)

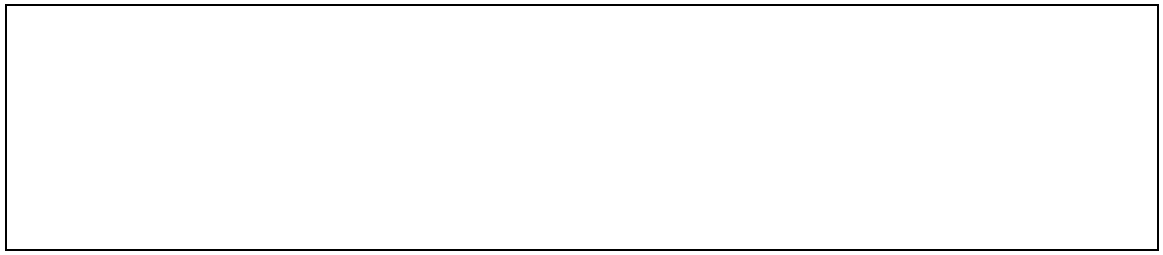
Make sure the server is running and execute attack 6:

```
$ java ch.zhaw.securitylab.publicfilestorage.PublicFileStorageTester
  localhost 6
```

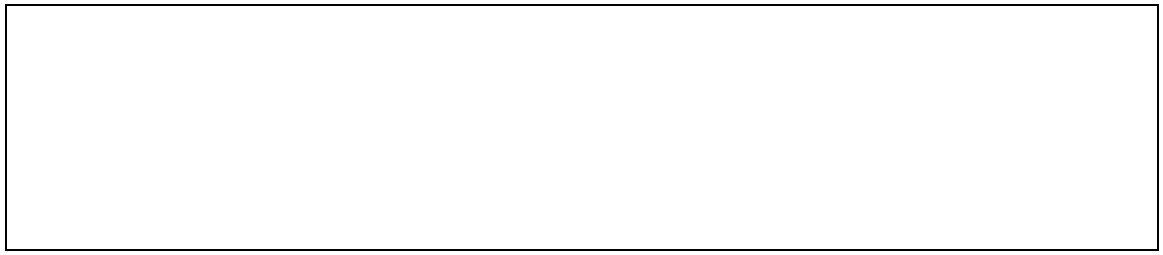
This should give access to file */etc/passwd* on the server. Note that other files on the server can also be read provided that the running server process has the necessary rights to do this. This is obviously a

bad vulnerability as it may provide an attacker with access to files that contain possible sensitive information. The attacker could also, for instance, access the *.java* files of the running server application to analyze the code for further vulnerabilities or to sell the code to others (assuming it were a valuable application).

Analyze the request and the source code of the server and explain why it is possible to get the file.



How would you fix this problem? Just think about a possible solution, don't implement anything yet.



The best way to fix this is again with input validation. Basically, you have to make sure that the filename specified in the *GET* request does not contain the slash (*/*) character, so it's not possible to escape «upwards» from the directory where the files are stored (here: *files/*).

Implement this by first providing a method *validateFilename* that gets the filename received in a *GET* request and that checks that the filename is legitimate. Do this again with a regular expression. In our case, assume a legitimate filename has a length between 1 and 100 characters and allowed characters are *a-z*, *A-Z*, *0-9*, and the following special characters: _+=$%?,.;:. Specifying the number of characters in a regular expression can be done with curly brackets that follow a character set (which is defined in square brackets). For instance, *[abcde]{5,10}* matches any sequence of lowercase characters a-e of length between 5 and 10 (including 5 and 10).

Then, use *validateFilename* right at the beginning of *serveFile* to validate the received filename. If it is not valid, respond with *NOK*. Also, do the same check at the beginning of *storeFile*, so an attacker cannot (over-)write arbitrary files on the system.

To verify whether your code modifications have the desired effect, execute the attack again and check the output of the test program. The attack should now fail. Also, repeat test 0 to make sure the server still works as it should when the client uses legitimate requests.

The attack discussed in this subsection is also called directory traversal attack as the attacker traverses the directory tree to access areas that shouldn't be accessible by him. In addition, the validation of the filename you just added has two beneficial side effects:

- As the */* character can no longer be used, it is enforced that a user cannot create or read files in subdirectories *files/*. This was specified Section 3, but so far, it was not enforced in the code.
- Attack 1 also makes use of a directory traversal attack. This means that with the newly added validation of the filename, attack 1 should not be possible anymore, even if the server runs with root permissions. Verify this by repeating attack 1 – it should fail.

6.7 Attack 7: Getting a file that shouldn’t be accessible (variant 2)

Make sure the server is running and execute attack 7:

```
$ java ch.zhaw.securitylab.publicfilestorage.PublicFileStorageTester
  localhost 7
```

This should again give access to file */etc/passwd* on the server. Hmm... didn't we just prevent this when discussing attack 6? So apparently, there's (at least) one additional vulnerability still present to achieve this attack goal.

Analyze the request and the source code of the server and explain why it is possible to get the file.

This attack shows that input validation is not always as easy as it seems because although we made sure in the previous subsection that filenames with slashes (*/*) are not accepted, directory traversal attacks are apparently still possible.

How would you fix this problem? Just think about a possible solution, don't implement anything yet.

This attack shows that supporting encoding schemes (here: URL encoding) is potentially dangerous as it may allow attacks that are often overlooked during development. The easy fix here would simply be to not support URL encoding at all, but let's assume that this is an important and mandatory feature in the file storage server that cannot be ignored. So let's implement it in a secure way.

Basically, the secure approach to deal with encoding schemes is to do the decoding first and do all further security checks on the decoded data. Therefore, first remove all calls of *urlDecode* in methods *serveFile*, *storeFile* and *executeSystemCommand*., as these calls are currently done after the data has been validated. Then, use *urlDecode* in *processRequest* right when extracting the argument from the first line of the request:

```
int indexSpace = line.indexOf(' ');
String requestType = line.substring(0, indexSpace);
String argument = urlDecode(line.substring(indexSpace + 1));
```

This guarantees that from now on, we are working with the raw decoded data and no longer have to worry about potentially encoded characters that may circumvent subsequent security checks.

To verify whether your code modifications have the desired effect, execute the attack again and check the output of the test program. The attack should now fail. Also, repeat test 0 to make sure the server still works as it should when the client uses legitimate requests.

6.8 Attack 8: Do a portscan on the server

Make sure the server is running and execute attack 8:

```
$ java ch.zhaw.securitylab.publicfilestorage.PublicFileStorageTester
  localhost 8
```

As a result of this, you first get the disk space (output of the command *du*) used by the files, followed by the output of a portscan (with *nmap*) that was done locally on the server system. A portscan is just used as an example here, other commands that are available on the server system could be used as well (e.g., to download and install malware).

Analyze the request and the source code of the server and explain why it is possible to do the portscan.

Apparently, the attack makes it possible to execute basically arbitrary commands on the server system and that's definitely another major vulnerability. This kind of attack is often identified as command injection.

Obviously, this is another case for input validation. Therefore, implement a method *validateCommand* that makes sure only valid commands can be used by the clients. Up to now and according to section 3.3, the only valid command is *USAGE* and it either needs * or . as an option. This can easily be enforced by using a regular expression. Once implemented, use *validateCommand* right at the beginning of method *executeSystemCommand*. If validation fails, respond with *NOK*.

To verify whether your code modifications have the desired effect, execute the attack again and check the output of the test program. The attack should now fail. Also, repeat test 0 to make sure the server still works as it should when the client uses legitimate requests.

6.9 Check that all tests work and that all attacks are prevented

As a final check, run *PublicFileStorageTester* without a test or attack number:

```
$ java ch.zhaw.securitylab.publicfilestorage.PublicFileStorageTester
  localhost
```

Inspect the output in detail. All tests in the beginning (test 0) should succeed and all attacks (attacks 1-8) should fail. If that's the case: congratulations, you have significantly improved the security of the file storage service! If that's not the case, correct the code until everything works as expected.

Lab Points

In this lab, you can get **2 Lab Points**. To get them, you must demonstrate that you successfully fixed the security vulnerabilities in the server program. This means that when running *PublicFileStorageTester* without a test or attack number (as described in section 6.9), test 0 should still work and attacks 1-8 should be prevented. Furthermore, you must provide reasonable answers to the questions in section 6 (write them directly into the boxes or into a separate document). Also, if requested by the instructor, you have to describe the changes you did in the source code to fix the vulnerabilities. Handing in your solution can be done in two different ways:

- Show and explain your answers and demonstrate your solution to the instructor in classroom, during the official lab lessons.
- Create screenshots or make a video of a terminal that shows the correctly running *PublicFileStorageTester* program and send this together with your answers to the questions in section 6 by e-

mail to the instructor. Use *SecLab - Public File Storage Service - group X - name1 name2* as the e-mail subject, corresponding to your group number and the names of the group members. If the video is too big for an e-mail, place it somewhere (e.g., SWITCHdrive) and include a link to it in the e-mail.